Programación Funcional Avanzada

Programación Dinámica Funcional

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Programación Dinámica

Calcular una vez, compartir varias

- Convertir un problema de programación en una recurrencia – calcularla almacenando valores intermedios en una colección.
- En lenguajes imperativos la colección es un arreglo – intercambiar espacio por tiempo eliminando redundancia.
- Calcular una vez y compartirlo varias veces – *exactamente* lo que hace la evaluación perezosa.

Se venden en combos. . .

- Se pueden comprar tequeños en combos de 6, 9 y 20.
- Se pueden comprar cero o más combos.

¿Se pueden comprar *exactamente* `N` tequeños?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Puedo comprar *exactamente* `N`?

Como un problema de decisión

```
compraR0 n = r!n
    where r    = listArray (0,n)
                           (True : map f [1..n])
          f i = i >=  6 && r!(i-6) ||
                i >=  9 && r!(i-9) ||
                i >= 20 && r!(i-20)
```

- $n-$ésima posición del arreglo `r` indica si se puede o no.
- Si puedo comprar $i-6$ o $i-9$ o $i-20$ tequeños, entonces puedo comprar i.
- La posición inicial es `True`.
- Arreglo *inmutable* se genera con una iteración cerrada.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Puedo comprar *exactamente* N?

Como un problema de decisión

```
comprar0 n = r!n
    where r    = listArray (0,n)
                           (True : map f [1..n])
          f i = i >=  6 && r!(i-6) ||
                i >=  9 && r!(i-9) ||
                i >= 20 && r!(i-20)
```

- $n-$ésima posición del arreglo `r` indica si se puede o no.
- Si puedo comprar $i-6$ o $i-9$ o $i-20$ tequeños, entonces puedo comprar i.
- La posición inicial es `True`.
- Arreglo *inmutable* se genera con una iteración cerrada.

Si pero, ¿*cuántos* combos necesito?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Puedo comprar *exactamente* N?

Como un problema de combinación

```
comprar1 n = r!n
  where r = listArray (0,n)
                      (Just (0,0,0) : map f [1..n])
        f i = case attempt (i-6) of
          Just (x,y,z) -> Just (x+1,y,z)
          _            -> case attempt (i-9) of
             Just (x,y,z) -> Just (x,y+1,z)
             _            -> case attempt (i-20) of
                Just (x,y,z) -> Just (x,y,z+1)
                _            -> Nothing
        attempt x = if x>=0 then r!x
                            else Nothing
```

- Arreglo indica `Maybe (Int,Int,Int)` – cuántos combos de 6, 9 y 20.

¿Puedo comprar *exactamente* N?

Como un problema de combinación

```
comprar1 n = r!n
  where r = listArray (0,n)
                      (Just (0,0,0) : map f [1..n])
        f i = case attempt (i-6) of
          Just (x,y,z) -> Just (x+1,y,z)
          _            -> case attempt (i-9) of
             Just (x,y,z) -> Just (x,y+1,z)
             _            -> case attempt (i-20) of
                Just (x,y,z) -> Just (x,y,z+1)
                _            -> Nothing
        attempt x = if x>=0 then r!x
                            else Nothing
```

- Arreglo indica `Maybe (Int,Int,Int)` – cuántos combos de 6, 9 y 20.
- Esa cascada de `case` apesta a Monad `Maybe`.

Una solución más regular e idiomática

```
comprarM n = r!n
  where r = listArray (0,n)
                      (Just (0,0,0) : map f [1..n])
        f i = do (x,y,z) <- attempt (i-6)
                    return (x+1,y,z)
              `mplus`
              do (x,y,z) <- attempt (i-9)
                    return (x,y+1,z)
              `mplus`
              do (x,y,z) <- attempt (i-20)
                    return (x,y,z+1)
        attempt x = guard (x>=0) >> r!x
```

- `Maybe` es `Monad` – `attempt` usa `guard` para "corto-circuito".
- `Maybe` es `MonadPlus` – `mplus` produce el primer resultado o `Nothing`

UNIVERSIDAD SIMÓN BOLÍVAR

No estamos siendo muy funcionales...

- Estamos manteniendo *todo* el arreglo en memoria – ¿qué va a pasar cuando `n` sea grande?
- Sería más eficiente si pudiéramos *reciclar* las posiciones bajas del arreglo una vez que no hacen falta.
- Eso lo hace el recolector de basura si realizamos los cómputos de manera que los valores previos no hagan falta.

Un viejo truco que siempre está vigente

Una lista que se consume a si misma

```
comprar2 n = go n (True : replicate 19 False)
  where go 0 cs = cs !! 0
        go n cs = go (n-1)
                         ((cs !! 5 ||
                           cs !! 8 ||
                           cs !! 19) : take 19 cs)
```

- Nunca necesitamos más de 20 elementos.
- El primero siempre es `True` y "sembramos" el resto con `False`.

Y ahora bajamos de nivel...

```
comprar3 n = go n 1
  where
    go :: Int -> Int -> Bool
    go 0 cs = odd cs
    go n cs = go (n-1) ((cs `shiftL` 1) .|.
                        if cs .&. 0x80120 /= 0 then 1
                                                else 0)
```

- $80120_{16} = 10000000000100100000_2$ – bits 20, 9 y 6.
- El arreglo está empaquetado en un mapa de bits – `Data.Bits`

¿Cuál es más rápido?

Usemos `Criterion`

Función	Promedio
comprar0	167.7562 μs
comprar1	192.1358 μs
comprarM	190.4110 μs
comprar2	663.5192 μs
comprar3	008.2330 μs

¿Cuál es más rápido?

Usemos `Criterion`

Función	Promedio
comprar0	167.7562 μs
comprar1	192.1358 μs
comprarM	190.4110 μs
comprar2	663.5192 μs
comprar3	008.2330 μs

¿Cuánto cuesta hacerlo así de rápido en C?

UNIVERSIDAD SIMÓN BOLÍVAR

Distancia de Levenshtein

Usado en procesamiento de lenguaje natural

- Medida de la diferencia entre dos secuencias – usualmente cadenas.
- Número mínimo de ediciones (agregar, eliminar o sustituir) para convertir una cadena en la otra.
- La distancia entre `maduro` y `balurdo` es tres
  - `Maduro` pasa a `Baduro` – sustitución.
  - `baDuro` pasa a `baLuro` – sustitucion.
  - `baluro` pasa a `balurDo` – inserción.
  - No hay manera de hacerlo en menos de tres ediciones.

¿Cómo se calcula?

Distancia de Levenshtein

Una recurrencia

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{cuando } min(i,j) = 0 \\ min & \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+[a_i \neq b_j] \end{cases} \end{cases}$$

- a y b son las cadenas a comparar – distancia es $lev_{a,b}(|a|,|b|)$
- La implantación recursiva es obvia e ineficiente.
- La implantación dinámica típica usa una matriz para conservar las distancias entre *todos* los prefijos.

UNIVERSIDAD SIMÓN BOLÍVAR

La matriz de "maduro" a "balurdo"

Guardando las distancias...

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{cuando } min(i,j) = 0 \\ min & \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+[a_i \neq b_j] \end{cases} \end{cases}$$

		m	a	d	u	r	o
	0	1	2	3	4	5	6
b	1	1	2	3	4	5	6
a	2	2	1	2	3	4	5
l	3	3	2	2	3	4	5
u	4	4	3	3	2	3	4
r	5	5	4	4	3	2	3
d	6	6	5	4	4	3	3
o	7	7	6	5	5	4	3

UNIVERSIDAD SIMÓN BOLÍVAR

Expresado recursivamente

Se presta para listas por comprensión

```
lev0 s t = d !! (length s) !! (length t)
  where
    d = [
          [ delta m n | n <- [0 .. length t] ]
          | m <- [0 .. length s]
        ]
    delta i 0 = i
    delta 0 j = j
    delta i j = minimum [
        d !! (i-1) !! j + 1,
        d !! i     !! (j-1) + 1,
        d !! (i-1) !! (j-1) +
                      (if s!!(i-1)==t!!(j-1) then 0
                                                else 1) ]
```

Con arreglos ha de ser más rápido. . .

Con arreglos ha de ser más rápido

- Imitar la solución imperativa requiere arreglos mutables.
  - Todos los cómputos son puros – operar en el Monad `ST`.
  - Datos *unboxed* para máximo desempeño.

Parece un trabajo para `STUArray`

Arreglos mutables en Monad ST

La inicialización. . .

```
lev1 s t = d ! (ls , lt)
  where s' = array (0,ls)
                   [ (i,x) | (i,x) <- zip [0..] s ]
                   :: UArray Int Char
        t' = array (0,lt)
                   [ (i,x) | (i,x) <- zip [0..] t ]
                   :: UArray Int Char
        ls = length s
        lt = length t
        (l,h) = ((0,0),(length s,length t))
```

- `s'` y `t'` – acceso rápido al $i-$ésimo caracter de cada palabra.
- `l` y `h` – coordenadas mínima y máxima del arreglo mutable.

UNIVERSIDAD SIMÓN BOLÍVAR

Arreglos mutables en Monad ST

La transformación...

```
d = runSTUArray $ do
    m <- newArray (l,h) 0
         :: ST s (STUArray s (Int,Int) Int)
    forM_ [0..ls] $ \i -> writeArray m (i,0) i
    forM_ [0..lt] $ \j -> writeArray m (0,j) j
    forM_ [1..lt] $ \j -> do
      forM_ [1..ls] $ \i -> do
        let c = if s'!(i-1) == t'!(j-1) then 0 else 1
        x <- readArray m (i-1,j)
        y <- readArray m (i,j-1)
        z <- readArray m (i-1,j-1)
        writeArray m (i,j) $ minimum [x+1, y+1, z+c]
    return m
```

- Arreglo *unboxed* de dos dimensiones – no nos importa el estado.
- Se inicializan los “márgenes”.
- Iteración monádica anidada – traducción del método iterativo.

¿Y es más rápido?

Otra vez `Criterion`

```
benchmarking lev0
[...]
mean: 6.826855 us,
      lb 6.803381 us, ub 6.857277 us, ci 0.950

benchmarking lev1
[...]
mean: 2.909211 us,
      lb 2.901831 us, ub 2.917778 us, ci 0.950
```

Poco más del doble de rápido

Construyendo recursivamente

Reconsideremos el método...

- El arreglo se construye en **una** pasada.
- Cada celda se construye en base a vecinas previas – su valor no cambiará más adelante.
- En realidad, el arreglo es *inmutable* una vez construido.
- Sirve un vulgar (y puro) `Data.Array` – siempre y cuando se construya recursivamente.
- Cada celda es una *función* de sus vecinas – y la vecindad es una función de la coordenada actual.

```
mkArray f bs = array bs
                     [ (i, f i) | i <- range bs ]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo recursivamente

Lazy functional epicness

```
lev2 sa sb = table ! (length sa, length sb)
  where
    arrA = listArray (0, length sa - 1) sa
    arrB = listArray (0, length sb - 1) sb
    table = mkArray f ((0,0), (length sa, length sb))
    f (ia, 0) = ia
    f (0 ,ib) = ib
    f (ia,ib)
      | a == b    = table ! (ia-1,ib-1)
      | otherwise = 1 + minimum [ table ! x |
                                  x <- [ (ia-1,ib-1),
                                         (ia-1,ib),
                                         (ia,ib-1)] ]
      where
        a = arrA ! (ia - 1)
        b = arrB ! (ib - 1)
```

¿Y es más rápido?

Otra vez Criterion

```
benchmarking lev0
[...]
mean: 6.826855 us,
      lb 6.803381 us, ub 6.857277 us, ci 0.950

benchmarking lev1
[...]
mean: 2.909211 us,
      lb 2.901831 us, ub 2.917778 us, ci 0.950

benchmarking lev2
[...]
mean: 2.222751 us,
      lb 2.216678 us, ub 2.230146 us, ci 0.950
```

Aún más rápido.

Quiero saber más...

- Documentación sobre Data.MemoTrie – Lazy Memoization using Tries
- Página sobre la Distancia de Levenshtein en WikiPedia

UNIVERSIDAD SIMÓN BOLÍVAR